# DYNAMIC PROGRAMMING

# Dynamic Programming

- **Dynamic Programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure (array, map,etc). Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup. So the next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time. This technique of storing solutions to subproblems instead of recomputing them is called memoization.

# Continue…

- “Programming” here means “planning”
- Main idea:

- set up a recurrence relating a solution to a larger instance to

solutions of some smaller instances

- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

# Elements of Dynamic Programming (DP)

**DP** is used to solve problems with the following characteristics:

- Simple subproblems

– We should be able to break the original problem to smaller subproblems that have the same structure

- Optimal substructure of the problems

– The optimal solution to the problem contains within optimal solutions to its subproblems.

- Overlapping sub-problems

– there exist some places where we solve the same subproblem more than once.

# Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution

# Principle of Optimality

- The dynamic Programming works on a principle of optimality.
- Principle of optimality states that in an optimal sequence of decisions or choices, each sub sequences must also be optimal.

# Example 1: Fibonacci numbers

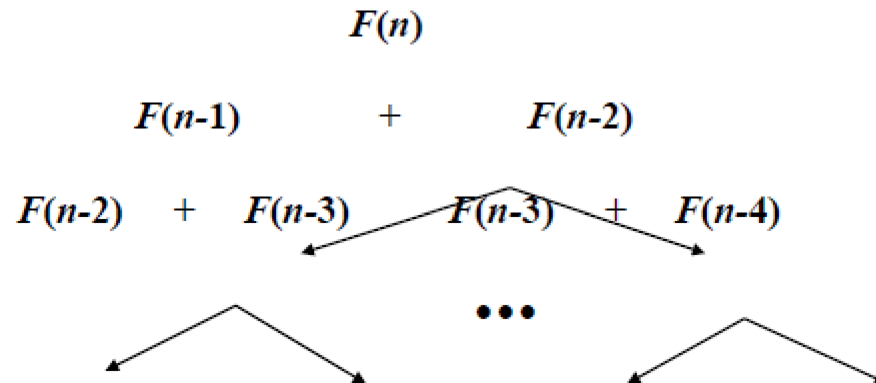
- Recall definition of Fibonacci numbers:

F(n) = F(n-1) + F(n-2)

F(0) = 0

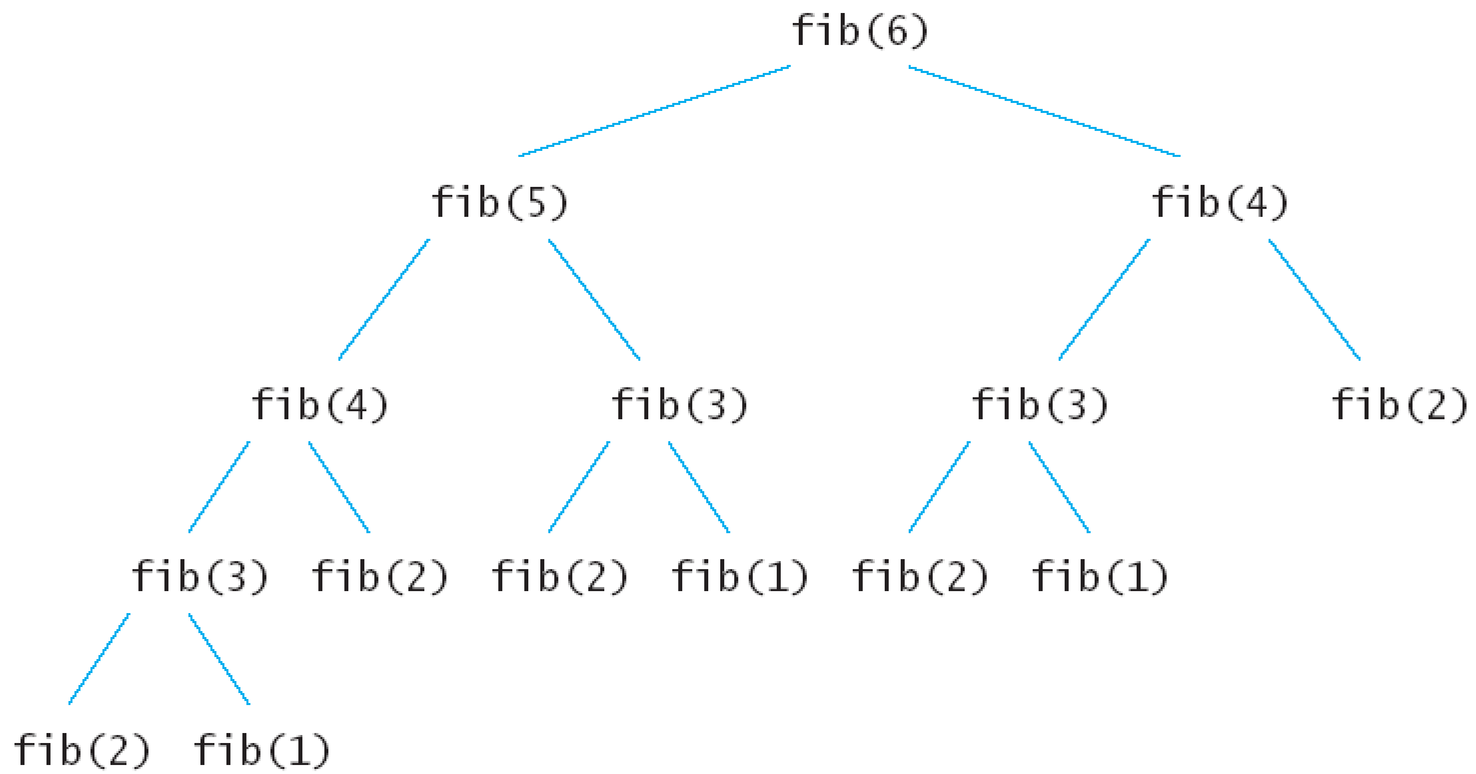
F(1) = 1

- Computing the nth Fibonacci number recursively (top-down):



# Fibonacci Numbers

- $F_n = F_{n-1} + F_{n-2}$    $n \geq 2$
- $F_0 = 0$, $F_1 = 1$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- Straightforward recursive procedure is slow!
- Let's draw the recursion tree


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(2)
fib(1)

# Continue…

- We can calculate Fn in linear time by remembering solutions to the solved subproblems – **dynamic programming**
- Compute solution in a **bottom-up** fashion
- In this case, only two values need to be remembered at any time

```
Fibonacci(n)
    F0←0
    F1←1
    for i ← 2 to n do
        Fi ← Fi-1 + Fi-2
```

# Knapsack Problem

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.
The following examples will establish our statement.

**Example-1**

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

| Item | A | B | C | D |
|---|---|---|---|---|
| Profit | 24 | 18 | 18 | 10 |
| Weight | 24 | 10 | 10 | 7 |

Without considering the profit per unit weight (pi/wi), if we apply Greedy approach to solve this problem, first item A will be selected as it will contribute maximum profit among all the elements.

After selecting item A, no more item will be selected. Hence, for this given set of items total profit is 24. Whereas, the optimal solution can be achieved by selecting items, B and C, where the total profit is 18 + 18 = 36.

## Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio pi/wi. Let us consider that the capacity of the knapsack is W = 60 and the items are as shown in the following table.

| Item | A | B | C |
| --- | --- | --- | --- |
| Price | 100 | 280 | 120 |
| Weight | 10 | 40 | 20 |
| Ratio | 10 | 7 | 6 |

Using the Greedy approach, first item A is selected. Then, the next item B is chosen. Hence, the total profit is 100 + 280 = 380. However, the optimal solution of this instance can be achieved by selecting items, B and C, where the total profit is 280 + 120 = 400.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

## Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

$W = 11$

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

# Dynamic Programming : False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

# Knapsack Problem : Bottom up

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

# Dynamic Programming : Adding a new variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

# Knapsack Algorithm

W + 1 →

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| φ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| { 1, 2, 3, 4, 5 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 34 | 40 |

n + 1 ↓

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

W = 11

OPT: { 4, 3 }
value = 22 + 18 = 40

# Knapsack Problem : Running Time

Running time. O (n W)

How much storage is needed?

The table we used above is of size O(n W).

For example, if W = 1000000 and n = 100, we need a total of 400 Mbytes ($10^6$ x 100 x 4 bytes).

This is too high.

The space requirement can be reduced: once row j has been computed, we don't need rows j – 1 and below.

Space reduces to 2W = 8 Mbytes in the above case.